VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Diagnostické zařízení pro zařízení měřicí tlak

Diagnostic Device for Pressure Measuring Equipment

2020 Jaromír Gąsior

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Zadání bakalářské práce

Student: **Jaromír Gąsior**

Studijní program: B0714A150001 Řídicí a informační systémy

Téma: Diagnostické zařízení pro zařízení měřicí tlak
Diagnostic Device for Pressure Measuring Equipment

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Analýza současného stavu měřicího řetězce.
2. Prostředky pro tvorbu diagnostické aplikace.
3. Sběrnice CAN.
4. Připojení měřicího řetězce (sběrnice CAN) k osobnímu počítači.
5. Návrh a implementace diagnostické aplikace v jazyce C#.
6. Testování vytvořené aplikace.
7. Závěrečná zhodnocení.

Seznam doporučené odborné literatury:

[1] VOSS, Wilifried. *A comprehensible guide to controller area network.* Greenfield: Cooperhill Technologies Corporation, 2005. ISBN 0-9765116-0-6.
[2] CHRISTIAN, Nagel. *Professional c# 7 and .net core 2.0.* Indiana, IN: John Wiley, 2018. ISBN-13: 978-1119449270.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jaromír Konečný, Ph.D.**

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA * ID16

doc. Ing. Jiří Koziorek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 14. května 2020 .....

Rád bych na tomto místě poděkoval vedoucímu své bakalářské práce, panu Ing. Jaromíru Konečnému, Ph.D. za věcné rady a připomínky k mé práci.

Abstrakt

Cílem této práce je návrh a implementace diagnostického zařízení pro zařízení měřicí tlak. V této práci jsou nejprve rozebrány možné prostředky pro výrobu diagnostické aplikace. Dále je zde popsána komunikační sběrnice CAN, na které dané zařízení komunikuje. Nakonec je zde popsán návrh a implementace jak připojení měřicího řetězce k osobnímu počítači, tak samotné diagnostické aplikace.

Klíčová slova: Bakalářská práce, CAN, UART, Diagnostické zařízení, Tlak, C, C#, XAML, WPF

Abstract

The aim of this thesis is development and implementation of the diagnostic device for pressure measuring equipment. In this thesis are first discussed possible means for production of the diagnostic application. Next point is the description of the CAN bus at which the equipment communicates. Finally, the development and implementation of connecting the measuring chain to the personal computer is described as well as the diagnostic application itself.

Keywords: Bachelor thesis, CAN, UART, Diagnostic device, Pressure, C, C#, XAML, WPF

Obsah

Seznam použitých zkratek a symbolů

CAN	–	Controller Area Network
CRC	–	Cyclic Redundancy Check
IR	–	InfraRed (Infračervené záření)
JSON	–	JavaScript Object Notation
MVVM	–	Model-View-ViewModel
NRZ	–	Non-Return to Zero
RAM	–	Random Access Memory
UART	–	Universal Asynchronous Receiver-Transmitter
USB	–	Universal Serial Bus
WPF	–	Windows Presentation Foundation
XAML	–	Extensible Application Markup Language
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

S rostoucími nároky na rychlejší a bezpečnější přenos dat v automobilech musel vzniknout i lepší řídicí systém mezi jednotlivými periferiemi. Proto firma Bosch vyvinula Controller Area Network, neboli CAN, což je sériová datová sběrnice pro využití především v automobilech. Postupem času byl CAN zdokonalován a začal se využívat ve stále více průmyslových odvětvích. Bylo tomu tak především kvůli velké flexibilitě použití, vysoké přesnosti, nízké ceně a vysoké přenosové rychlosti.

Cílem této práce je vytvoření diagnostického zařízení pro komunikační sběrnici CAN, jež je součástí zařízení měřicí tlak. Diagnostické zařízení se skládá z komunikačního uzlu sběrnice CAN a diagnostické aplikace napsané v jazyce C#.

Komunikační uzel sběrnice CAN slouží převážně jako převodník mezi komunikačním protokolem CAN a osobním počítačem. Diagnostická aplikace na osobním počítači umožňuje uživateli sledovat aktuální dění na sběrnici a případně taky ovládat chování sběrnice odesláním zprávy.

Práce je rozdělena na teoretickou a praktickou část. Teoretická část obsahuje aktuální stav měřicího řetězce, možné prostředky pro tvorbu uživatelské aplikace a obecnou definici sběrnice CAN. Praktická část se pak skládá z realizace propojení měřicího řetězce k osobnímu počítači, vytvoření uživatelské aplikace pro diagnostiku a testování vytvořeného zařízení.

Na začátku práce, v kapitole 2, je popsán současný stav měřicího řetězce. Kapitola obsahuje blokové schéma měřicího řetězce spolu s vysvětlením funkčnosti, významu a způsobu komunikace jeho jednotlivých částí.

V kapitole 3 se nachází stručný přehled programovacích jazyků, které lze použít k vytvoření diagnostické aplikace na osobním počítači. Jsou zde vytaženy hlavní výhody a nevýhody vybraných programovacích jazyků a rozdíly mezi nimi. Vybrány byly jedny z nejznámějších a nejpoužívanějších programovacích jazyků vhodných pro tvorbu diagnostických aplikací, a to C++, Java, C# a Python. Všechny tyto jazyky jsou zcela zdarma a volně šiřitelné.

V kapitole 4 této práce je vysvětlen princip fungování sběrnice CAN. Na začátku této kapitoly se nachází stručná historie vzniku sběrnice CAN. Dále je popsána fyzická realizace komunikačních uzlů na sběrnici spolu s linkovou a fyzickou vrstvou jednotlivých zpráv. V závěru této kapitoly je možné nalézt sběrnicový komunikační protokol, tedy jednotlivé typy a obsah zpráv, vyskytujících se na sběrnici.

Kapitola 5 se zabývá propojením měřicího řetězce s osobním počítačem. To je důležité zejména proto, že osobní počítač není schopen fungovat jako komunikační uzel na sběrnici CAN, tedy nedokáže číst zprávy přímo ze sběrnice. K tomu je využit speciálně upravený komunikační uzel, který funguje jako převodník zpráv mezi sběrnicí a osobním počítačem. Umožňuje převod zpráv ze sběrnicového CAN protokolu na komunikaci přes UART, tedy přes počítačový sériový port. V této kapitole lze nalézt i kódy psané v jazyce C, které jsou nahrány v komunikačním uzlu.

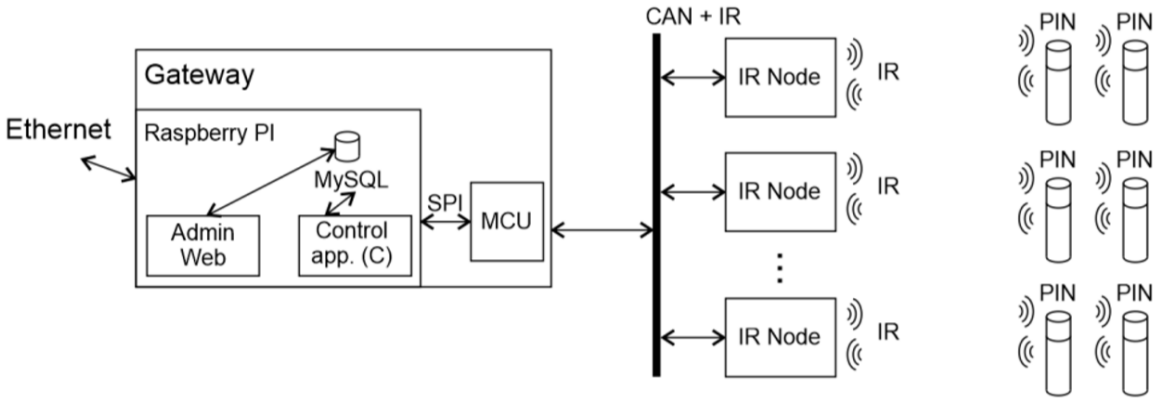
Návrh a implementace samotné diagnostické aplikace v jazyce C# se nachází v kapitole 6. Jako první bod této kapitoly je popsán WPF framework, na kterém aplikace běží. Zbytek kapitoly se věnuje postupu při tvoření aplikace a jejímu konečnému řešení. Aplikace umožňuje koncovému uživateli oboustrannou komunikaci s měřicím řetězcem. Kapitola obsahuje ukázky jak kódu, tak i vizuálního prostředí. Na konci kapitoly jsou popsány možnosti aplikace a způsob jejího ovládání.

Testování správného chodu diagnostického zařízení se nachází v kapitole 7. Je zde popsána schopnost zařízení přenášet bezchybně jednotlivé zprávy ze sběrnice CAN do diagnostické aplikace na osobním počítači a naopak. Dále je v této kapitole popsán způsob ošetření výjimek a chybových stavů a taky použitý logovací systém.

Poslední kapitola 8 je věnována závěrečnému zhodnocení vykonané práce.

2 Analýza současného stavu měřicího řetězce

Blokové schéma měřicího řetězce daného zařízení pro měření tlaku je vyobrazeno na obrázku 1. Celý systém sestává ze 3 hlavních částí, a to zprava z jednotlivých chytrých měřicích kolíků, z komunikačních uzlů a z Gateway, neboli Brány. [1]



Obrázek 1: Blokové schéma měřicího řetězce

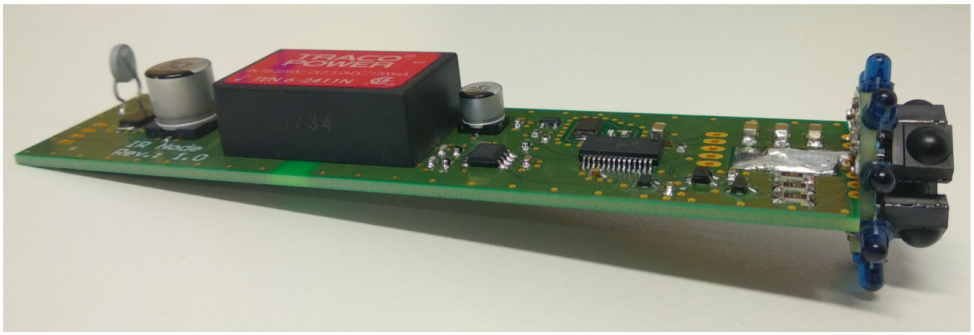
2.1 Chytré měřicí kolíky

Chytré kolíky přímo měří tlak pomocí kapacitních snímačů. Naměřená data ukládají na svou vnitřní paměť RAM a následně data vysílají jako neviditelné IR záření do svého okolí. Kolík disponuje IR modulem, který umožňuje příjem a vysílání pomocí IR záření.

2.2 Spojovací uzly

Spojovací uzly umožňují na jedné straně komunikaci s měřicími kolíky a komunikaci s bránou na straně druhé. Uvnitř uzlu se uskutečňuje skládání dat a kontrola správnosti a celistvosti dat. Fyzická realizace spojovacího uzlu je na obrázku 2. Spojovací uzly jsou umístěny ve velmi blízkém okolí měřicích kolíků. [1]

Komunikace s měřicími kolíky je uskutečněna pomocí kruhového terčíku na jednom konci uzlu, který slouží jako přijímač i jako vysílač IR zpráv. Jedním z požadavků bylo, aby byl uzel schopný přijímat zprávy ze všech stran a vysílat zprávy do všech stran. Proto jsou terčíky uzlů kruhového tvaru. Jako přijímače používá uzel klasické IR přijímače a jako vysílače používá uzel klasické 3mm LED diody pracující v infračervené oblasti záření. Uzly a chytré kolíky mezi sebou komunikují na základě přiloženého IR protokolu. [1]



Obrázek 2: Spojovací uzel měřicího řetězce [1]

Na druhé straně spojovací uzel posílá data na sběrnici CAN. Na této sběrnici jsou propojeny všechny uzly a brána, která zde funguje jako nadřazený systém. Sběrnice CAN je zároveň multiplexována pro synchronizované vysílání zpráv směrem ke kolíkům. Přenos dat po IR je realizován fyzickou vrstvou sběrnice CAN. Mikrokontrolér uvnitř uzlu deaktivuje CAN modul v době, kdy vysílá data na IR vysílač. Proto je možné přenášet data, která nesplňují standardní CAN protokol. Komunikace je zde tedy řízena na základě vlastního CAN protokolu a mikrokontrolér upravuje data do podoby vhodné pro přenos. [1]

2.3 Brána

Brána sbírá a hlavně řídí informace ze systému. Sestává ze dvou bloků, a to z platformy Raspberry Pi a z mikrokontroléru řídící tok dat. Komunikace mezi nimi je řízena komunikačním protokolem SPI. Mimo to zajišťuje taky buffer pro příchozí data ze sběrnice CAN.

Platforma Raspberry Pi zde funguje jako soubor několika webových technologií, tvořících řídící aplikační celek. Brána pracuje s databází MySQL, která obsahuje především konfiguraci zařízení, ale taky naměřená data. Dále brána disponuje administrační webovou aplikací, jež umožňuje konfiguraci brány, či jednotlivých komunikačních uzlů. Pro propojení webových technologií s hardwarem je zde ještě řídící aplikace napsaná v jazyce C. [2]

3 Prostředky pro tvorbu diagnostické aplikace

3.1 C++

C++ vydal v roce 1986 Bjarne Stroustrup v Bellových laboratořích jako objektové rozšíření jazyka C. Vzhledem ke svému věku je C++ historicky jeden z nejvíce používaných jazyků na světě. Programy napsané v jazyce C++ jsou přenositelné na různé platformy. Napsané části programu se kompilují na objektové soubory, které se následně pomocí knihoven propojí v jeden spustitelný soubor. [3, 4, 5]

C++ je univerzální programovací jazyk, který podporuje objektově orientované, procedurální a generické programování. C++ je ze všech programovacích jazyků nejspíše nejvíce flexibilní. Je tomu tak zejména kvůli možnosti přístupu do low-level hardwarových funkcí. Dobře napsaný program v jazyce C++ bude vždy nejrychlejší a nejméně náročný na paměť. [4, 5, 6]

Největší výhodou jazyka C++ je, že poskytuje programátorovi plnou kontrolu nad chováním jak programu tak vyhrazenou pamětí. Proto může být programování v tomto jazyce jak velmi jednoduché, tak velmi složité. Tento jazyk zkrátka neudělá nic za vás. [5]

Zásadním krokem oproti jazyku C je využití vlastností objektově orientovaného programování. Tím se myslí zejména zapouzdření, dědičnost a polymorfismus. Zapouzdření znamená, že aplikace využívá objekty spojené s daty a metodami, které se nazývají třídy. Třída se dokáže skrýt před zbytkem programu a být přístupná pouze svému majiteli. Definujeme tedy takzvaná oprávnění k přístupu. Dědičnost umožňuje programu odvozovat nové třídy z jedné nebo více již existujících tříd, přebírat jejich vlastnosti a doplňovat je novými. Tímto můžeme vytvářet takzvanou stromovou hierarchii tříd. Polymorfismus, neboli vícetvarost, umožňuje volání stejného objektu s různými parametry. [4]

3.2 Java

Java poprvé spatřila světlo světa v roce 1995. Vytvořil ji James Gosling ve firmě Sun Microsystems, kterou od roku 2010 vlastní společnost Oracle Corporation. V dnešní době je Java považována za jeden z nejrozšířenějších a nejvíce používaných programovacích jazyků na světě. [3, 7, 8]

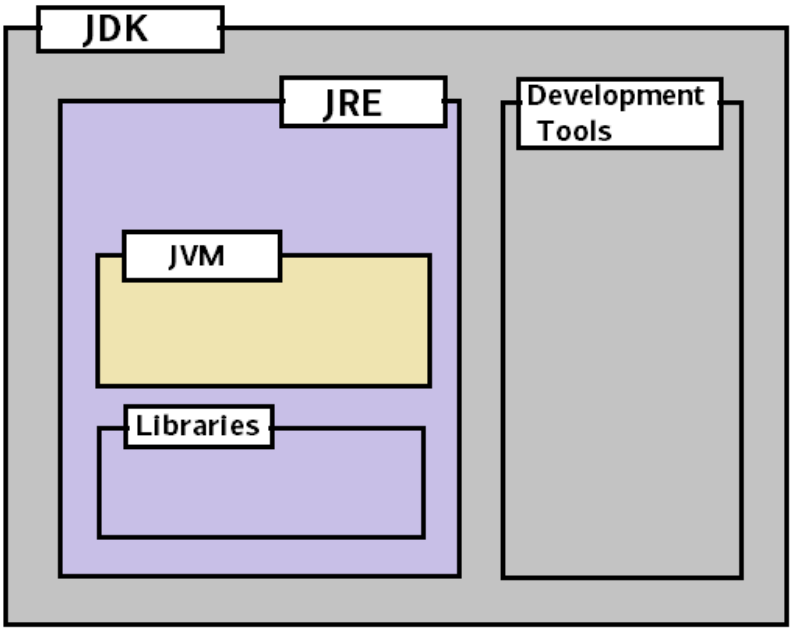
Jedná se o objektově orientovaný, dynamický, přenositelný a velmi jednoduchý programovací jazyk. Jendoduchý ve smyslu psaní kódu v porovnání s jazyky C nebo C++, z nichž Java vychází. Syntakticky je velmi podobný jazyku C#. Jazyk Java je zároveň vhodný jak pro tvorbu počítačových aplikací, tak pro tvorbu webových aplikací, mobilních aplikací, databází, serverů

nebo i her. [7, 8]

Java je poskytována zcela zdarma a mezi její největší výhody patří multiplatformost a přenositelnost. To znamená, že programy napsané v Javě jsou spustitelné prakticky na jakémkoliv zařízení, například na osobním počítači s libovolným operačním systémem, na mobilním telefonu, na různých zabudovaných zařízeních nebo třeba na čipových kartách. Kód psaný v Javě se v první řadě kompiluje na mezikód. Ten je následně spouštěn na JVM (Java Virtual Machine — Virtuální Stroj Javy), který lze jednoduše nainstalovat na libovolném zařízení. Proto stačí napsat jeden jediný kód, jenž je následně spustitelný na libovolném zařízení, které má nainstalované JVM. Kvůli tomuto chování se říká, že Java je jazyk „interpretovaný“. [7, 8]

Tento přístup způsobuje, že výsledný program by mohl v konečné fázi být pomalejší než kód napsaný v jazyce C nebo C++. Pro kompenzaci rychlosti běhu programu vznikl JIT (Just In Time Compiler — Just In Time Kompilace), který překládá mezikód do strojového kódu procesoru. To běh aplikace zrychluje, ale výrazně zvyšuje nároky na paměť. [7, 8]

Správu paměti zabezpečuje automatický Garbage Collector. Díky němu programátor objekty pouze vytváří a nestará se o uvolňování paměti. Jakmile zmizí všechny odkazy na daný objekt, Garbage Collector paměť uvolní. [7, 8]



Obrázek 3: Distribuční balíčky Javy [8]

Firma Oracle distribuuje Javu ve dvou různých balíčcích: [8]

1. Balíček JRE (Java Runtime Environment) — Určený pro koncového zákazníka. Obsahuje JVM a knihovny pro spuštění Java programů.

2. Balíček JDK (Java Development Kit) — Určený pro vývojáře aplikací. Jeho součástí je JRE a vývojové nástroje pro tvorbu aplikací (Development Tools).

3.3 C#

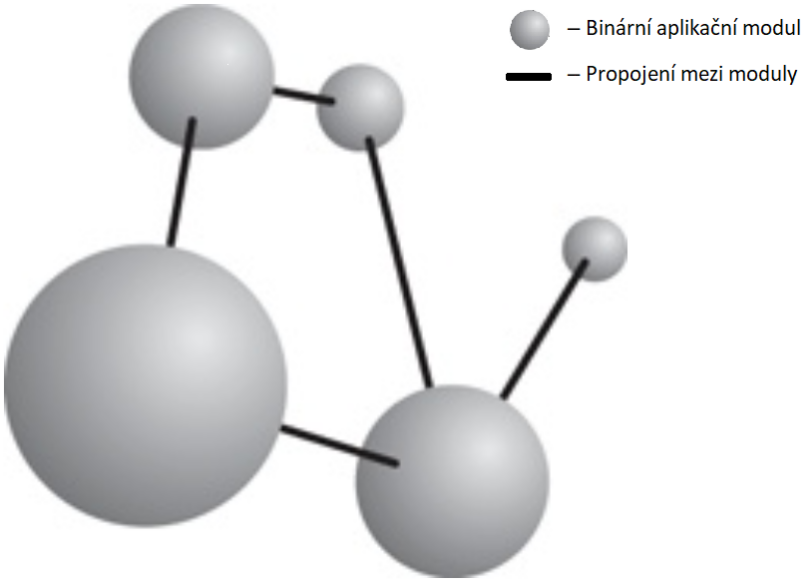
Programovací jazyk C# byl vydán v roce 2002 firmou Microsoft v čele s Andersem Hejlsbergem pro použití ve frameworku .NET. C# v mnohém vychází z jazyků C++, Pascal a Java. C# je nejenom objektově orientovaný, ale hlavně komponentně orientovaný programovací jazyk, který podporuje události (events), vlastnosti (properties), atributy (attributes) a vytváření sestav (assemblies). Neustále se vyvíjí a objevují se stále nové knihovny a funkce. [9]

S myšlenkou komponentně orientovaného jazyka přisel právě programovací jazyk C#. Rozdíl mezi objektově orientovaným a komponentně orientovaným programováním je v interpretaci tříd, neboli v tom, jak se díváme na konečnou spustitelnou aplikaci. Jazyk C# nám umožňuje použít od začátku takzvaný „prvotně třídový koncept“. [9, 10]

Objektově orientované programování funguje na základě vztahů mezi jednotlivými třídami, které se po kompilaci spojují do jednoho binárního spustitelného souboru. Všechny třídy sdílí ve výsledku stejný paměťový prostor, stejný proces, stejnou adresu a stejná přístupová práva. Při následné změně jedné třídy může tedy dojít k poškození programu. [10]

Komponentně orientované programování naopak funguje jako soubor jednotlivých binárních aplikačních modulů a propojení mezi nimi (viz obrázek 4). Jednotlivé moduly jsou kompilovány i spouštěny nezávisle na sobě, aniž by programátor musel znát jejich obsah. Při změně obsahu jedné třídy se nemusí kompilovat znovu celý program, ale jen daná třída. Změna je možná i za běhu aplikace a to pouze pod podmínkou, že zrovna daná třída není používána. Po rekompilaci třídy za běhu aplikace je okamžitě možné ji kýmkoli v programu použít. [10]

.Net framework kompiluje kódy podobně jako Java. Kompilaci programu řídí CLR (Compilation Language Runtime), které je součástí .NET framework, takže funguje pro všechny programovací jazyky v něm obsažené. CLR převádí zdrojový kód na IL (Intermediate Language) kód, který je schopný sám otevírat soubory knihoven DLL (Dynamic-Link Library) nebo přímo spustitelné EXE soubory (Executable), obsahující kód .NET frameworku. Součástí CLR je i JIT (Just In Time) kompilátor, který generuje nativní kód, když je program spuštěn. V neposlední řadě obsahuje CLR i Garbage Collector zodpovědný za uvolňování nepoužívané paměti. [9]



Obrázek 4: Model komponentně orientovaného programování [10]

Díky komponentně orientovanému programování vznikly v jazyce C# věci jako částečné třídy, statické třídy, virtuální metody, skryté metody, abstraktní třídy a metody, zapečetěné třídy, lambda výrazy a mnoho dalšího. [9]

3.4 Python

Programovací jazyk Python byl oficiálně vydán roku 1991. Jeho tvůrcem byl holanďan Guido van Rossum. Název Python vznikl ze zábavného pořadu Monty Python's Flying Circus. Rossum tvrdí, že toto jméno je vhodné proto, že i podle tvůrců pořadu nemá slovo žádný smysl. Python dodnes nemá žádné oficiální logo. [11, 12]

Python je populární zejména proto, že je velmi snadné se ho naučit a zakládá si na přehlednosti a čitelnosti zdrojového kódu. Jedná se o interpretovaný, interaktivní, víceúrovňový a objektově orientovaný programovací jazyk, který je dostupný na většině platforem. Python je vhodný zejména pro rychlý vývoj aplikací v mnoha oblastech nebo na takzvané „skriptování", čímž se rozumí tvorba skriptů pro webové aplikace. [3, 11]

Pod pojmem interaktivní programovací jazyk se myslí, že můžeme okamžitě vidět výsledky toho, co vytvoříme. To je výhodné zejména pro webové aplikace nebo pro jednoduché konzolové či okenní aplikace. [11, 12]

Že je Python jazyk víceúrovňový vychází z toho, že jeho interpret lze dynamicky doplňovat o nové funkce a datové typy psané v jazycích C nebo C++. Funguje tedy jako nízkoúrovňový

programovací jazyk, který může pracovat i se samotnou pamětí, jako je C++, nebo jako víceúrovňový jazyk, jako třeba Java nebo C#, a nebo obojí. [11]

Kódy v jazyce Python lze psát prakticky v čemkoliv. Existuje sice mnoho speciálních volně šiřitelných Integrovaných vývojářských prostředí pro psaní v jazyce Python, ale narozdíl od ostatních programovacích jazyků není pro vývoj aplikací v tomto jazyce potřeba. To proto, že syntaxe tohoto jazyka je jiná oproti ostatním programovacím jazykům. [11, 12]

Jak již bylo řečeno, Python je založen na přehlednosti a čitelnosti kódu. Jeho úseky jsou přehledně rozděleny pomocí striktně daných odsazení. Ty potom nahrazují klasické složené závorky, které můžeme vidět v ostatních programovacích jazycích. [12]

4 Sběrnice CAN

4.1 Obecná definice sběrnice CAN

CAN je sériová, asynchronní, multi-master a plně duplexní datová sběrnice vyvinutá firmou Bosch. [13]

4.1.1 Historie

V roce 1986 vydala oficiálně firma Bosch technické specifikace ke sběrnici CAN. Finální verze, označována jako CAN 2.0, byla vydána v roce 1991. Specifikace CAN 2.0 obsahuje dvě části. Část A, označována jako CAN 2.0A nebo taky CAN standard, a část B, označována jako CAN 2.0B nebo taky CAN extended. Stal se standardem ISO 11898 v roce 1993. O něco později byl tento standard rozdělen na ISO 11898-1, popisující linkovou vrstvu přenosu, ISO 11898-2, popisující fyzickou vrstvu vysoko-rychlostního CAN (angl. „high-speed CAN“), a ISO 11898-3, popisující fyzickou vrstvu nízko-rychlostního CAN (angl. „low-speed CAN“). [14, 15]

4.1.2 CAN uzly

Sběrnice slouží pro propojení tzv. uzlů (angl. „node“). Aby sběrnice fungovala, musí obsahovat minimálně dva tyto uzly. Složitost jednoho uzlu se velmi liší. Může fungovat jako jednoduché vstupně/výstupní zařízení, nebo taky může sloužit jako vestavěný počítač s rozhraním CAN a sofistikovaným softwarem. Ve speciálních případech může uzel sloužit i jako komunikační brána mezi sítí CAN a počítačem. [15]

Každý uzel, připojený k této sběrnici, slyší všechny přenosy, takže není možné poslat zprávu jen jednomu přijímači. Platí tedy, že každý vysílač je taky přijímač. Každý uzel musí obsahovat:

- Centrální procesorovou jednotku (CPU), mikroprocesor nebo hostitelský procesor
- Řadič CAN
- Budič CAN

Procesor rozhoduje, co příchozí zprávy znamenají a jaké zprávy chce odeslat na sběrnici.

Řadič CAN bývá zpravidla součástí mikrokontroléru. Při přijímání zprávy uchovává jednotlivé bity, dokud není zpráva celá a pak ji pošle procesoru ke zpracování. Při odesílání pošle procesor zprávu řadiči a ten, pokud je sběrnice volná, posílá seriově jednotlivé bity.

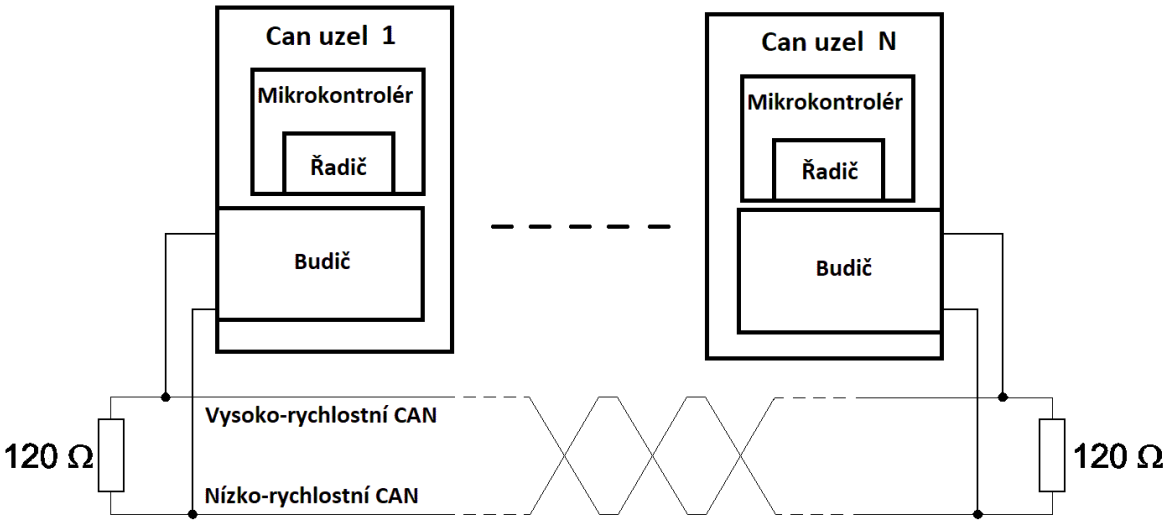
Budič CAN je vysílač a zároveň přijímač, jenž se nachází na výstupu z uzlu. Převádí napěťové úrovně mezi úrovní, na které funguje sběrnice, a úrovní, na které funguje mikrokontrolér, v obou směrech. [14]

4.2 CAN protokol

4.2.1 Fyzická vrstva

Všechny uzly jsou mezi sebou propojeny jako dvoudrátová sběrnice. Ta fyzicky vypadá jako zkroucený pár vysoko-rychlostního a nízko-rychlostního CAN s nominální charakteristickou impedancí kabelu 120 Ω. Vysoko-rychlostní CAN linka pak má 120 Ω rezistor na každém konci sběrnice a nízko-rychlostní CAN linka vyžaduje 120 Ω rezistor na každém uzlu. Odpor mezi těmito dvěma linkami by pak měl odpovídat 60 Ω. [13, 15]

Napěťové úrovně sběrnice CAN se označují jako dominantní a recesivní, přičemž dominantní je logická 0 a recesivní logická 1. Chtějí-li vysílat zprávu dva uzly ve stejnou chvíli, oba pozorují, co se na sběrnici děje. Aby nedošlo ke kolizi, platí, že pokud jeden z uzlů táhne sběrnici do logické 0 a jakýkoliv jiný uzel táhne sběrnici do logické 1, celá sběrnice je v logické 0. Vidí-li vysílající uzel, že vysílá logickou 1 a na sběrnici vidí logickou 0, sám přestane vysílat. Po dokončení transakce se pokusí uzel, který se vzdal vysílání, odeslat zprávu na sběrnici znovu. Tímto se na sběrnici pomocí identifikátoru určuje i priorita zprávy. [13]



Obrázek 5: Fyzická vrstva sběrnice CAN

4.2.2 Linková vrstva

Sběrnice CAN využívá tzv. vkládání bitů (angl. „Bit stuffing“) pro zajištění synchronizace rámce zprávy. Jde o vkládání nadbytečných bitů do rámce zprávy. Vždy po pěti bitech stejné polarity vloží odesílatel jeden bit opačné polarity. Příjemci zprávy následně tyto nadbytečné bity sami odstraní. [14]

To je důležité zejména kvůli tomu, že sběrnice CAN vyžívá pro bitovou reprezentaci unipolární NRZ kód. To znamená, že logická 1 je reprezentována jednou hodnotou napětí a logická 0 je reprezentována jinou hodnotou napětí stejné polarity, která nemusí být nulová. Neexistuje zde žádný třetí neutrální stav. [16]

Odesílaná data nemají žádnou adresu ale tzv. identifikátor (ID), jenž má každý uzel jedinečný. Tento identifikátor určuje jak obsah zprávy, tak i prioritu zprávy, přičemž čím menší je hodnota identifikátoru, tím vyšší je priorita zprávy. [14]

4.2.3 Rozdělení jednotlivých polí rámce zprávy podle linkové vrstvy

Linková vrstva definuje rámce CAN zprávy. Celý rámec se dělí na 7 hlavních polí (více v tabulkách 2 a 3 nebo na obrázcích 6 a 7):

- Startovací pole (Start of frame)
- Rozhodovací pole, nebo taky Arbitráž (Arbitration field)
- Kontrolní pole (Control field)
- Datové pole (Data field)
- Pole CRC (CRC field)
- Pole ACK (ACK field)
- Koncové pole (End of frame)

Startovací pole je jeden dominantní startovací bit označující zahájení transakce.

Rozhodovací pole, označováno taky jako arbitráž, určuje obsah a prioritu zprávy na sběrnici. U standardního formátu rámce zprávy se skládá z identifikátoru zprávy a RTR bitu určujícího, jestli se jedná o datový nebo vzdálený rámec. Pro rozšířený formát arbitráž obsahuje identifikátor A, recesivní SRR bit, IDE bit, jenž informuje sběrnici, zda se jedná o standardní nebo rozšířený formát, identifikátor B a RTR bit.

Kontrolní pole má u standardního formátu rámce IDE bit určující, zda se jedná o standardní nebo rozšířený formát rámce, jeden dominantní rezervovaný bit a datovou délku udávající počet bajtů datového pole. Rozšířený formát rámce obsahuje místo IDE bitu ještě jeden dominantní rezervovaný bit.

Datové pole je pole s variabilní velikostí 0–8 bajtů obsahující samotná data zprávy.

Pole CRC je bezpečnostní pole, které kontroluje bitové chyby v datovém poli zprávy. Počítá si ho každý uzel sám a porovnává s tímto polem.

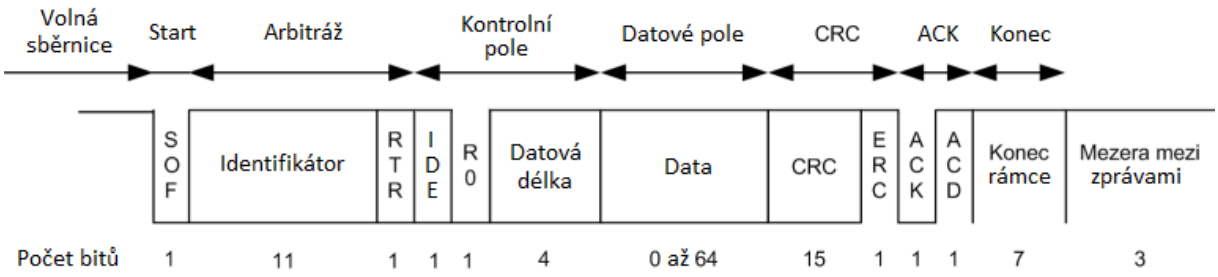
Pole ACK slouží jako potvrzovací pole o přijetí zprávy. Skládá se ze dvou bitů. ACK bit je při odeslání zprávy recesivní a při přijetí zprávy jakýmkoliv jiným uzlem se stává dominantním. Oddělovač ACK pak hlásí, nastane-li chyba ACK.

Koncové pole je pole sedmi bitů určujících konec transakce. [13, 14, 16]

4.2.4 Rozdělení formátů rámce zprávy podle linkové vrstvy

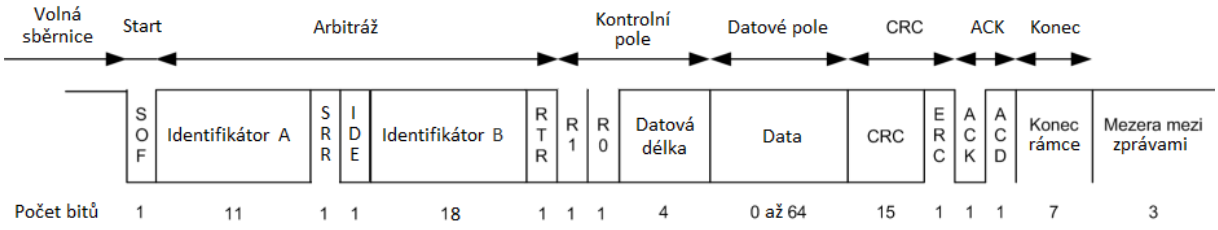
CAN protokol definuje 2 formáty datového rámce zprávy, lišící se především délkou identifikátoru (více v tabulkách 2 a 3): [13, 16]

1. Standardní formát (Standard format, CAN 2.0A), disponující 11 bitovým identifikátorem



Obrázek 6: Standardní formát datového rámce CAN zprávy

2. Prodloužený formát (Extended format, CAN 2.0B), diponující 29 bitovým identifikátorem



Obrázek 7: Rozšířený formát datového rámce CAN zprávy

4.2.5 Rozdělení typů rámce zprávy podle linkové vrstvy

Linková vrstva protokolu CAN definuje 4 typy rámců zpráv:

1. Datový rámec (Data frame) — Nejběžnější typ zprávy. Rámec musí obsahovat všech sedm hlavních polí.

2. Vzdálený rámec (Remote frame) — Vypadá stejně jako datový rámec jen se dvěma hlavními rozdíly:

   - Je explicitně označený jako vzdálený rámec (RTR bit v poli Arbitráže je recesivní místo dominantního).
   - Nemá žádné datové pole.

   Hlavní význam tohoto typu zprávy je žádost o data. To znamená, že pokud nějaký uzel žádá o zaslání dat, pošle vzdálený rámec s identifikátorem zprávy, kterou chce získat. Nějaký jiný uzel, vysílající zprávu s tímto identifikátorem, vyplní datové pole zprávy a tu znovu odešle. Aby toto fungovalo, uzel žádající o data musí předem vyplnit i datovou délku.

   Stane-li se, že se ve stejnou chvíli na sběrnici objeví datový rámec a vzdálený rámec se stejným identifikátorem, vyšší prioritu má vždy datový rámec právě kvůli dominantní úrovni RTR bitu v arbitráži zprávy.

   CAN standard ovšem nedefinuje přesné chování vzdáleného rámce. Může se stát, že toto chování je třeba samostatně nastavit, nebo třeba že sběrnice bude po přijetí vzdáleného rámce pouze informovat lokální CPU.

3. Chybový rámec (Error frame) — Porušuje veškeré chování CAN zpráv. Pokud uzel detekuje jakoukoliv chybu zprávy (chyba rámce, CRC chyba, chyba Bit stuffingu, bitová chyba...), odešle chybový rámec a tím donutí i všechny ostatní uzly vyslat chybový rámec. Poté se vysílající uzel pokusí zprávu odeslat znovu. Na sběrnici existuje propracovaný systém chybových čítaču znemožňující zacyklení chybových rámců.

   Chybový rámec obsahuje chybový příznak (Error flag), sestávající z 6 po sobě jdoucích stejných bitů (tímto porušuje pravidlo pro vkládání bitů), a chybový oddělovač (Error delimiter), jenž sestává z osmi recesivnívh bitů.

   Chybový příznak může mít dva tvary:

   - Aktivní — 6 po sobě jdoucích dominantních bitů
   - Pasivní — 6 po sobě jdoucích recesivních bitů

   Aktivní chybový příznak vysílá uzel, který chybu zachytil. Pasivní chybový příznak vysílají všechny ostatní uzly, když zjistí aktivní chybový příznak na sběrnici.

Chybový oddělovač poskytuje dostatek prostoru pro to, aby všechny uzly na sběrnici odeslali svůj chybový příznak.

4. Rámec přetížení (Overload frame) — Vysílá ho uzel, který není schopen dále vysílat nebo přijímat další zprávy. Vypadá stejně jako chybový rámec ale slouží pouze k oddálení vyslání další zprávy. Sestává z příznaku přetížení (Overload flag), obsahujícího šest dominantních bitů, a z oddělovače přetížení (Overload delimiter), obsahujícího osm recesivních bitů.

   Rámec přetížení se v současné době už téměř nepoužívá. Moderní sběrnice CAN jsou dost chytré a rychlé na to, aby jej nepotřebovaly použít. [13, 14, 16]

5 Připojení měřicího řetězce k osobnímu počítači

Osobní počítač neumí komunikovat se sběrnicí CAN přímo. Proto je třeba navrhnout něco jako převodník mezi komunikačním protokolem CAN a nějakým jiným komunikačním protokolem, se kterým už osobní počítač umí komunikovat. V tomto případě byl použit jeden z CAN uzlů, který na jedné straně komunikuje se sběrnicí CAN a na straně druhé s osobním počítačem přes rozhraní USB. Mikrokontrolér tohoto uzlu obsahuje program, jenž převádí zprávy z datového formátu protokolu CAN na datový formát sériového rozhraní UART a naopak. Kvůli skutečnosti, že je sběrnice CAN multiplexována pro přenos IR zpráv a funguje na základě vlastního nestandardního CAN protokolu, nemohl být použit standardní převodník CAN.

5.1 UART

UART není sběrnice jako třeba CAN. Je to fyzický elektronický obvod uvnitř mikrokontroléru, který pouze přijímá a vysílá zprávy vždy po jednom bajtu (tedy 8 bitů). K tomu mu stačí pouze dva kabely, a to Rx (Receiver — přijímač) a Tx (Transmitter — vysílač). UART funguje jako asynchronní sériová komunikace. [17]

Při nečinnosti komunikace UART udržuje na výstupu logickou 1. UART zpráva se zkládá z několika oblastí:

Tabulka 1: Struktura UART zprávy

Oblast	Bitová délka	Popis
Startovací bit	1	Logická 0 zahajující transakci.
Datová oblast	8	Data vysílané zprávy.
Paritní bit	1	Kontrolní bit.
Konec zprávy	1 nebo 2	Logická 1 označující konec transakce.

Není-li určeno jinak, UART posílá data od nejméně významného bitu (LSB = least significant bit) po nejvýznamější bit (MSB = most significant bit). [17]

UART na obou koncích komunikace sčítá počet logických jedniček v přenášené zprávě. Je-li počet jedniček sudý, paritní bit zprávy je v logické 0. Je-li počet jedniček lichý, paritní bit je v logické 1. Toto je jediný způsob, jak UART kontroluje bitové chyby zprávy. [17]

5.2 RFC 1662

Protože je přenos dat z uzlu do diagnostické aplikace uskutečněn jako asynchronní sériová komunikace, je třeba kvůli synchronizaci definovat komunikační protokol mezi těmito dvěma zařízeními. V tomto případě byl zvolen standard RFC 1662 s 32-bitovým kontrolním polem.

Kontrolní pole může být buď 16-bitové (2 bajty) nebo 32-bitové (4 bajty). Obsahuje kontrolní součet CRC32, který zaručuje, že při přenosu zprávy nedošlo k žádné bitové chybě. [18]

Protokol RFC 1662 definuje přenos jednotlivých bajtových zpráv do jednoho celku. Pro dosažení synchronizace v komunikaci používá jako startovací a ukončovací bajt 0x7E označovaný jako „FLAG“. Mezi jednotlivými zprávami stačí jen jeden FLAG, který pak slouží jako oddělovač jednotlivých zpráv. Jsou-li mezi zprávami dva za sebou jdoucí bajty 0x7E, protokol to považuje za prázdnou zprávu a sám ji smaže. [18]

Problém nastává, když chceme jako součást zprávy odeslat bajt 0x7E. Musíme tedy nahradit tento bajt jinou sekvencí dvou bajtů, označovanou jako „ESCAPE FLAG“, a to 0x7D a 0x5E. Tímto se vyřeší problém s odesláním datového bajtu 0x7E, ale nastává nový problém v podobě odeslání datového bajtu 0x7D. Ten musíme opět nahradit sekvencí dvou bajtů, označovanou jako „CONTROL ESCAPE“, a to 0x7D a 0x5D. Použitím těchto dvou sekvencí už nemůže nastat žádný problém v rozpoznání významu zprávy. Musí se ovšem vždy počítat s tím, že velikost přenášené zprávy není pevně daná právě kvůli tomuto zdvojování některých bajtů zprávy. [18]

5.3 Program v mikrokontroléru

5.3.1 Hlavní chod programu

Zprávy ze sběrnice CAN přicházejí jako ucelený balíček obsahující všechna data dané zprávy. Pro znázornění datového balíčku CAN jsou v programu definovány struktury jak pro tvar potřebný pro sběrnici CAN (1), tak pro tvar, který používá diagnostická aplikace na straně osobního počítače (2).

```
typedef struct {
    unsigned char dataLength;
    unsigned long ID;
    unsigned char data[8];
}CANPacket;
```

Výpis 1: Struktura CAN zprávy pro sběrnici

```
typedef struct {
    unsigned char type;
    CANPacket can_packet;
    unsigned char crc[4];
}CANMessage;
```

Výpis 2: Struktura CAN zprávy pro diagnostickou aplikaci

Aplikace používá pro převod zpráv 3 funkce. Jedna slouží pro převod komunikace z diagnostické aplikace do sběrnice (7), druhá pro opačnou komunikaci ze sběrnice do diagnostické aplikace (8) a třetí pro výpočet kontrolního součtu CRC32.

```
void CANToUART(CANPacket packet, unsigned char *UARTData);
CANMessage UARTToCAN(unsigned char *UARTData);
unsigned int crc32(unsigned char *message);
```

Výpis 3: Funkční bloky užité v mikrokontroléru

Samotný program mikrokontroléru pak obsahuje pouze kontrolu tří podmínek v nekonečném cyklu. První kontroluje obsah bufferu zpráv ze sběrnice CAN. Druhý kontroluje obsah bufferu zpráv z UARTu. Třetí pak kontroluje, jestli se v programu nachází nějaká kompletní zpráva přijatá z UARTu.

5.3.2 Převod z CAN na UART

Pro převod zprávy z protokolu CAN na komunikaci přes UART používá program v mikrokontroléru strukturu (2) a funkce (8) a CRC32.

Při příjmu zprávy ze sběrnice CAN nastane v programu přerušení, ve kterém se přijatá zpráva uloží do bufferu pro CAN zprávy. Program potom v normálním běhu zjistí, že buffer pro CAN zprávy není prázdný a spustí funkci (8).

Tato funkce postupně rozkládá zprávu na pole bajtů v pořadí, v jakém diagnostická aplikace zprávy přijímá. Zprávu následně zakóduje do formátu dodržující standard RFC 1662. To znamená, že nahradí každý bajt 0x7E sekvencí bajtů 0x7D a 0x5E, každý bajt 0x7D nahradí sekvencí bajtů 0x7D a 0x5D a na začátek a konec zprávy umístí bajt 0x7E.

Nakonec program cyklicky odesílá zprávu po jednotlivých bajtech přes UART do osobního počítače. Ten zprávu přijme přes sériový komunikační port a zprávu si sám dekóduje.

5.3.3 Převod z UART na CAN

Pro převod zprávy z UARTu na protokol CAN používá program v mikrokontroléru strukturu (1) a funkci (7).

Protože UART dokáže poslat jako zprávu vždy jen jeden bajt, nestačí jen sledovat, jestli buffer zpráv z UARTu není prázdný. Musí se vždy kontrolovat i obsah zprávy abychom věděli, zda už nastal konec zprávy.

Při příjmu zprávy z UARTu se vyvolá v programu přerušení s nejvyšší prioritou, v němž se uloží daná zpráva, tedy jeden bajt, do bufferu pro zprávy z UARTu. Hlavní program pak cyklicky čte jednotlivé zprávy z bufferu, dokud nenarazí na bajt 0x7E, tedy na ukončovací FLAG zprávy. Až konečný bajt najde, označí zprávu za kompletní a opouští cyklus čtení zpráv.

Ve chvíli, kdy program nalezne kompletní zprávu, tak ji dekóduje z formátu zprávy dodržující standard RFC 1662. To znamená, že program odstraní bajty 0x7E na začátku a na konci zprávy, pak nahradí každou sekvenci bajtů 0x7D a 0x5E bajtem 0x7E a každou sekvenci bajtů 0x7D a 0x5D bajtem 0x7D.

Nakonec program zabalí příslušnou zprávu jako strukturu CAN zprávy a označí zprávu jako připravenou k odeslání na sběrnici. Komunikační uzel se následně snaží danou zprávu odeslat na sběrnici.

6 Návrh a implementace diagnostické aplikace

Aplikace by měla být schopna číst aktuální dění na sběrnici CAN a přehledně zobrazovat jak data, tak jejich význam. Dále by měla být schopna umožnit uživateli rychlý a intuitivní výběr z přednastavených možností zaslání zprávy na sběrnici CAN, případně by měla umožnit uživateli snadno přidávat nové možnosti zaslaných zpráv. V neposlední řadě by pak aplikace měla umožňovat uživateli jak externí uložení všech nastavení a dat, tak externí načtení dříve nasbíraných dat. Samožřejmostí je ošetření všech výjimek a neočekávaných stavů.

6.1 WPF framework

Pro vytvoření samotné diagnostické aplikace byl zvolen programovací jazyk C# ve vývojovém prostředí Microsoft Visual Studio. Framework by zvolen WPF, což je soubor knihoven pro tvorbu formulářových aplikací a je zároveň součástí .NET frameworku. Jedná se o modernější verzi obecně známého frameworku Windows Forms. Hlavní výhodou frameworku WPF je využití návrhového vzoru MVVM. Důvodem pro zvolení WPF frameworku byla především snaha o vytvoření moderního, inteligentního a uživatelsky přívětivého prostředí.

6.1.1 Návrhový vzor MVVM

MVVM je návrhový vzor navržený speciálně pro aplikace, využívající WPF framework. Rozděluje program do jednotlivých vrstev, jež od sebe oddělují data, uživatelské rozhraní a stav apliakce. Jak napovídá název tohoto vzoru, jednotlivé vrstvy jsou:

- Model
- View
- ViewModel

Vrstva Model představuje data, se kterými program pracuje. Může se jednat například o pracovní databázi.

View je vrstva programu reprezentující uživatelské rozhraní psané v jazyce XAML. Jedná se o variaci jazyka XML, kterou zavedla společnost Microsoft pro vytváření uživatelských rozhraní svých aplikací. Psaní kódu v jazyce XAML tedy připomíná spíše psaní webových aplikací v jazyce HTML. Tato vrstva programu tedy představuje například okno aplikace, stránku nebo ovládací prvek. Někdy se tato vrstva taky označuje jako UI (User Interface - Uživatelské rozhraní), formulář nebo taky prezentační vrstva.

O propojení vrstev a celkovém chování programu se stará vrstva ViewModel. Jedná se o speciální třídu, jež si drží stav aplikace. S ovládacími prvky uživatelského rozhraní se propojuje

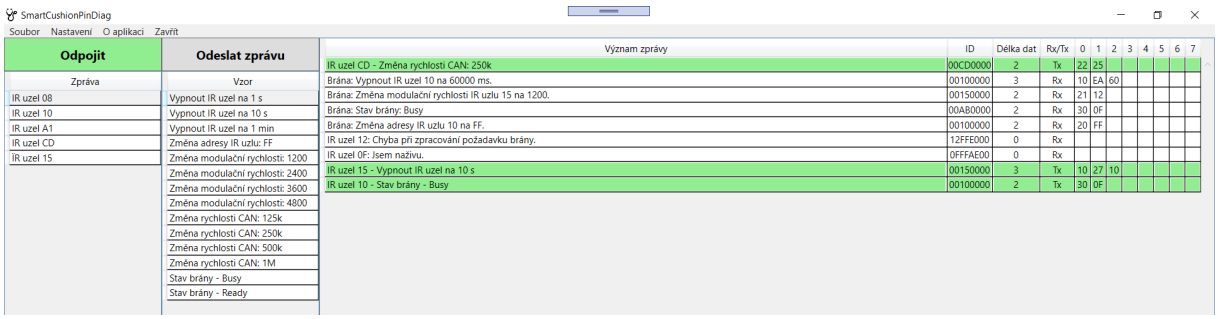
takzvanými „Bindingy“ a „Commandy“. Uživatelské rozhraní se dotazuje ViewModelu, jak má vykreslovat ovládací prvky a naopak přijetí jakéhokoliv požadavku od uživatele se okamžitě projeví na stavu aplikace. Zde je využíván programovací jazyk C#. [19, 20]

6.2 Návrh hlavního okna uživatelského rozhraní aplikace

Základní požadavek na tuto diagnostickou aplikaci byl, aby poskytovala informace o aktuálním dění na připojené sběrnici CAN a aby umožňovala základní ovládání této sběrnice. Ovládáním je myšleno odesílání předem definovaných zpráv podle přiloženého CAN protokolu.

6.2.1 Prvotní varianty uživatelského rozhraní

Při návrhu grafického uživatelského rozhraní aplikace byly zvažovány dvě varianty. Bylo tomu zejména kvůli tomu, jak má být aplikace ovládána.

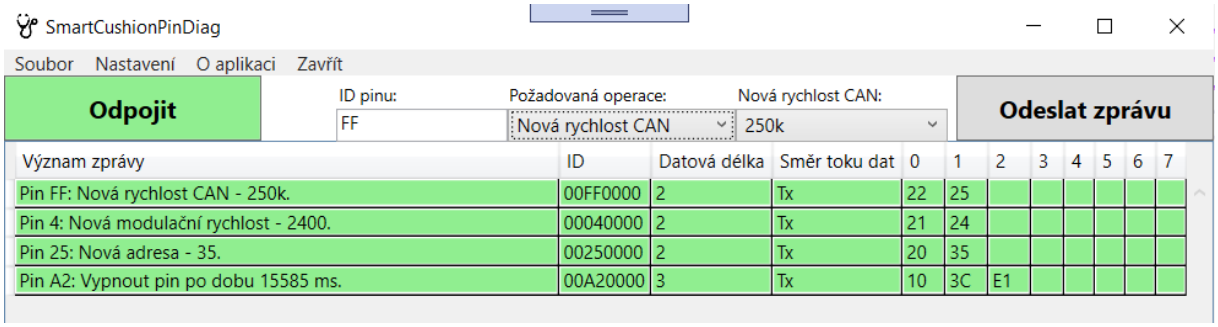


Obrázek 8: První varianta návrhu hlavního okna grafického uživatelského rozhraní

První variantu lze vidět na obrázku 8. Sestává z rozdělení hlavního okna do dvou částí. Větší část, umístěná vpravo, slouží k zobrazování jak přijatých, tak odeslaných datových zpráv, spolu se stručným popisem významu dané zprávy. Část druhá slouží k odeslání zprávy na sběrnici CAN. Uživatel si vybere z předem nadefinovaných zpráv, jež jsou všechny zobrazeny ve dvou DataGridech v levé části aplikace. V prvním z těchto DataGridů si uživatel zvolí zprávu, obsahující především identifikátor celé zprávy. Druhý DataGrid se dynamicky mění na základě zvolené zprávy. Zobrazuje datové vzory definované pro uživatelem zvolenou zprávu, které obsahují především data zprávy. Po vybrání zprávy a vzoru lze zprávu odeslat na sběrnici CAN.

Tato varianta se pro uživatele jeví rychlejší a vizuálně přívětivější. To ovšem platí pouze pro předem přesně nadefinované zprávy. Chce-li uživatel změnit třeba jen jedinou hodnotu v datech, musí v nastavení definovat nový vzor zprávy. Z toho vyplývá, že nelze rychle změnit význam odesílané zprávy, nebyl-li dříve definován.

Obrázek 9 zobrazuje druhou ze zvažovaných variant grafického uživatelského rozhraní. Je zde k dispozici pouze jeden DataGrid, zobrazující jak přijaté, tak odeslané datové zprávy, spolu se



Obrázek 9: Druhá varianta návrhu hlavního okna grafického uživatelského rozhraní

stručným popisem významu dané zprávy. Nad tímto DataGridem se nachází něco jako ovládací panel, který umožňuje uživateli odeslat zprávu. Tomu stačí jednoduše vepsat ID pinu, kam chce příkaz odeslat, následně si vybrat z předem nadefinovaných operací, které lze na pinu provést, a nakonec, podle zvolené operace, si buď zvolit z nadefinovaných možností pro tuto operaci nebo přímo napsat hodnotu, kterou chce odeslat. Pak lze takto uživatelem upravenou zprávu odeslat na sběrnici CAN.

Výhodou této varianty uživatelského rozhraní je především snadná a rychlá úprava odesílaných dat. Je třeba ale mít předem definovány všechny možné požadované operace. Tato varianta připadá v úvahu v případě, že je třeba často a hlavně rychle měnit obsah zprávy, kterou chce uživatel odeslat.

6.2.2 Finální verze grafického uživatelského rozhraní

Vzhledem k tomu, že nebyl kladen požadavek na editaci dat zprávy, byla zvolena první verze grafického uživatelského rozhraní, kterou lze vidět na obrázku 8. Pro usnadnění definování nových zpráv byla aplikace rozšířena alespoň o základní sadu 14 datových vzorů, která se vytvoří automaticky ke každé nově vytvořené zprávě. Jsou to nálsedující vzory:

- Vypnout IR uzel na 1 s
- Vypnout IR uzel na 10 s
- Vypnout IR uzel na 1 min
- Změna adresy IR uzlu: FF
- Změna modulační rychlosti: 1200
- Změna modulační rychlosti: 2400
- Změna modulační rychlosti: 3600
- Změna modulační rychlosti: 4800

- Změna rychlosti CAN: 125k
- Změna rychlosti CAN: 250k
- Změna rychlosti CAN: 500k
- Změna rychlosti CAN: 1M
- Stav brány: Busy
- Stav brány: Ready

V aplikaci lze libovolně vytvářet, měnit i mazat jakékoliv zprávy či vzory.

Pro správný chod aplikace se stačí připojit k sériovému portu. Lze tak učinit tlačítkem v levém horním rohu, a to po nastavení správného portu a modulační rychlosti. Je-li port odpojen, tlačítko je červeně zvýrazněné s nápisem „Připojit“. V opačném případě je tlačítko zvýrazněno zeleně s nápisem „Odpojit“. Zvýraznění tlačítka je zde proto, aby aplikace nějakým způsobem upozornila uživatele, že je aplikace buďto připravena na příjem zpráv ze sběrnice, anebo že je odpojena a čeká na připojení.

6.2.3 Vlastnosti hlavního okna aplikace

Data jsou ke všem DataGridům připojena pomocí Bindingů a jsou formátována pomocí jazyka XAML. Ukázku lze vidět ve výpisu 4.

```
<DataGridTextColumn Header="ID" Binding="{Binding id}" Width="auto"
    CanUserReorder="False" IsReadOnly="True">
    <DataGridTextColumn.ElementStyle>
        <Style TargetType="TextBlock">
            <Setter Property="TextAlignment" Value="Center"/>
        </Style>
    </DataGridTextColumn.ElementStyle>
</DataGridTextColumn>
```

Výpis 4: Ukázka použití jazyka XAML pro formátování sloupce zobrazujícího ID zprávy

Pro přehlednost výpisu zpráv ze sběrnice je třeba rozlišit příchozí zprávy od těch odchozích. K tomu je v aplikaci využito podmíněné formátování, o které se stará kód ve výpisu 5. Jedná-li se o odchozí zprávu z aplikace na sběrnici, zvýrazní se celý řádek s danou zprávou zelenou barvou. V opačném případě se zpráva zobrazí bez zvýraznění. Aplikace čte směr toku dat ze sloupce Rx/Tx.

```
<DataGrid.CellStyle>
    <Style TargetType="{x:Type DataGridCell}">
        <Style.Triggers>
            <DataTrigger Binding="{Binding com}" Value="Tx">
                <Setter Property="Background" Value="LightGreen" />
            </DataTrigger>
        </Style.Triggers>
    </Style>
</DataGrid.CellStyle>
```

Výpis 5: Podmíněné formátování zobrazované zprávy v jazyce XAML

Jak již bylo zmíněno dříve, aplikace zobrazuje pro každou zprávu význam dané zprávy. Ten je zpracováván jinak pro příchozí zprávy a jinak pro ty odchozí. Jako význam odchozí zprávy se pouze poskládá dohromady název odesílané zprávy a jejího zvoleného vzoru. Při skládání významu příchozí zprávy ze sběrnice využívá aplikace speciální funkci, která skládá význam zprávy jak podle identifikátoru zprávy, tak podle dat dané zprávy. Význam zpráv je zpracováván podle přiloženého sběrnicového CAN protokolu.

6.3 Menu aplikace

Aplikace disponuje jednoduchým menu, jehož některé položky mají určené i klávesové zkratky. Všechny položky menu jsou řízeny pomocí Commandů, jejichž příklad je zobrazen ve výpisu zdrojového kódu 6. Položky tohoto menu i s jejich klávesovými zkratkami jsou:

- Soubor
  - Nový (CTRL + N)
  - Uložit jako
  - Uložit (CTRL + S)
  - Načíst (CTRL + O)
- Nastavení (CTRL + P)
- O aplikaci
- Zavřít (CTRL + Q)

Položka menu s názvem nový umožňuje uživateli smazat veškerá zobrazovaná data, tedy přijaté a odeslané zprávy. Aplikace se uživatele vždy zeptá, zda chce opravdu data nenávratně smazat, aby nedošlo k nechtěné ztrátě dat.

Uložit a Uložit jako jsou standardní položky menu pro ukládání dat. Všechny přijaté a odeslané zprávy se serializují do formátu JSON a uloží do externího souboru. Položka Uložit jako je přístupná vždy a zobrazí uživateli dialogové okno pro zvolení vlastní cesty, kam chce data uložit. Položka Uložit ale nemůže být přístupná, nezná-li aplikace předem umístění, kam má data uložit. V takovém případě je tedy pro uživatele zablokována. Uvolní se pro uživatele v případě, že někdy dříve uložil soubor pomocí položky Uložit jako, anebo používá dříve uložený soubor s daty pomocí položky Načíst. Aplikace pak pracuje s tímto datovým souborem a položkou Uložit přepisuje zvolený soubor. Klávesová zkratka CTRL + S aktivuje položku Uložit, zná-li cestu k souboru. V opačném případě aktivuje položku Uložit jako a umožní uživateli zvolit cestu k souboru.

Položka Načíst používá stejný formát dat, tedy JSON. Aktivováním této položky se uživateli zobrazí klasické dialogové okno pro zvolení cesty k datovým souborům s koncovkou json. Po zvolení souboru se smažou aktuální zobrazovaná data a nahradí se dříve uloženými daty ze souboru.

Nastavení je speciální položka menu, sloužící k ovládání aplikace. Aktivací této položky se otevře nové okno s nastavením aplikace. Hlavní okno aplikace potom není přístupné, dokud uživatel nedokončí práci v okně s nastavením a nezavře toto okno. Možnosti nastavení jsou detailně popsány dále v kapitole 6.4.

O aplikaci je položka menu, která otevře nové okno mimo aplikaci, v němž lze nalézt základní informace o aplikaci.

Ukončení celé aplikace je možné buď klasicky křížkem v pravém horním rohu nebo použitím položky menu s názvem Zavřít, případně ještě použitím klávesové zkratky CTRL + Q.

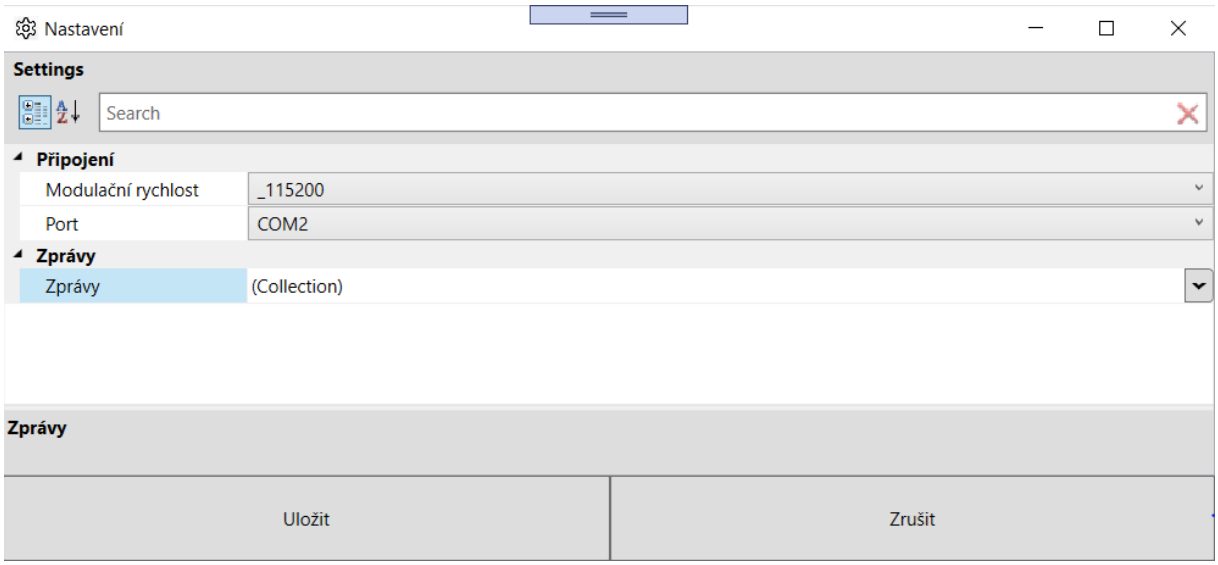
```
<Window.Resources>
    <RoutedUICommand x:Key="New" Text="Nový">
        <RoutedUICommand.InputGestures>
            <KeyGesture>CTRL+N</KeyGesture>
        </RoutedUICommand.InputGestures>
    </RoutedUICommand>
</Window.Resources>
<Window.CommandBindings>
    <CommandBinding Command="{StaticResource New}" Executed="
        NewCommandBinding_Executed"/>
</Window.CommandBindings>
<Menu HorizontalAlignment="Stretch" Height="17" VerticalAlignment="Top" Width="
    Auto">
```

```
    <MenuItem Command="{StaticResource New}" />
</Menu>
```

Výpis 6: Příklad propojování vlastností prvků aplikace pomocí commandů v jazyce XAML

6.4 Možnosti nastavení

Nastavení je okno, ve kterém probíhá ovládání chování celé aplikace. Toto okno lze vidět na obrázku 10. Uživatel se do nastavení dostane kliknutím na položku menu s názvem Natavení, nebo použitím klávesové zkratky CTRL + P. Po otevření okna je uživateli umožněna práce pouze v tomto okně. Aplikace v pozadí stále poběží normálním způsobem.

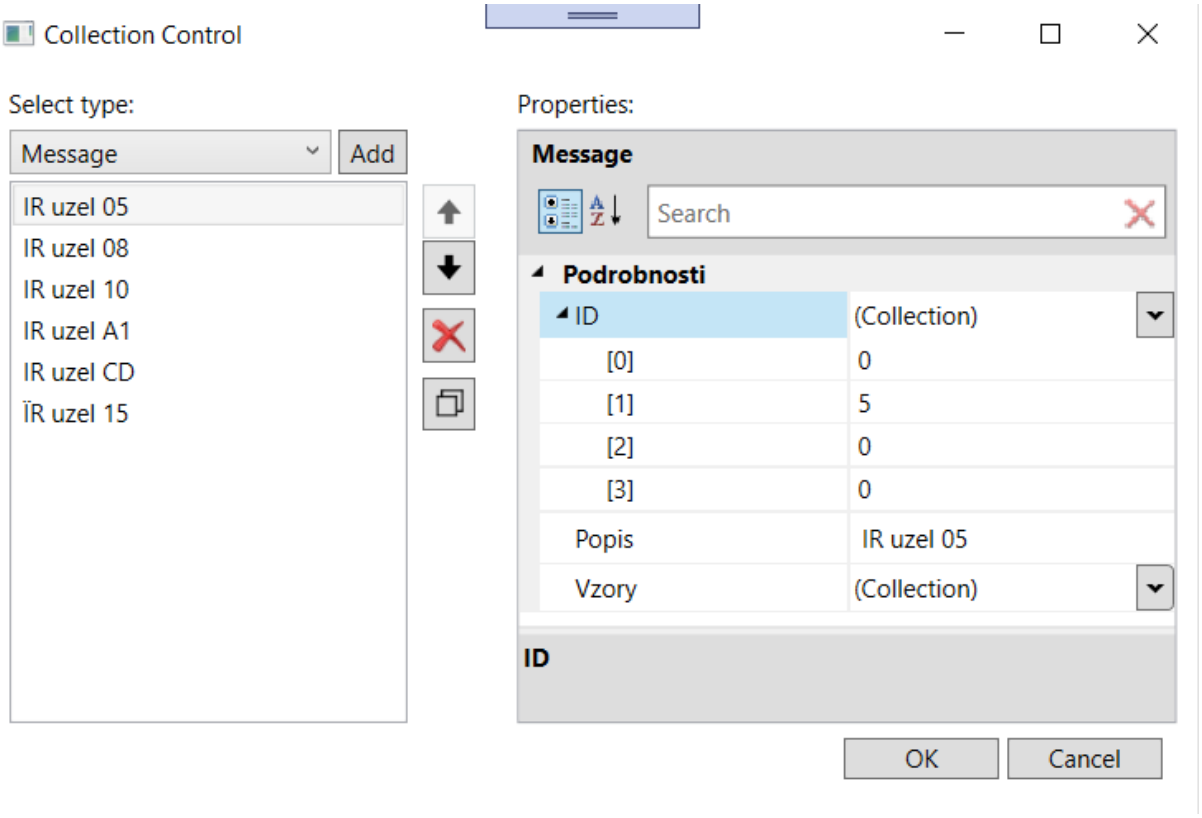


Obrázek 10: Okno nastavení

Požadavek na ovládání byl za využití ovládacího prvku jazyka C# PropertyGrid. WPF ale normálně tento prvek neobsahuje. Proto bylo zapotřebí doinstalovat „Extended WPF toolkit“ od společnosti Exceed. Jedná se o balíček rozšířených vlastností a více ovládacích prvků pro aplikace využívající WPF framework. Součástí tohoto balíčku je i požadovaný ovládací prvek PropertyGrid.

Diagnostické zařízení komunikuje přes počítačový USB port. Součástí nastavení je zvolení portu a modulační rychlosti sériového portu, kterému odpovídá připojený USB port.

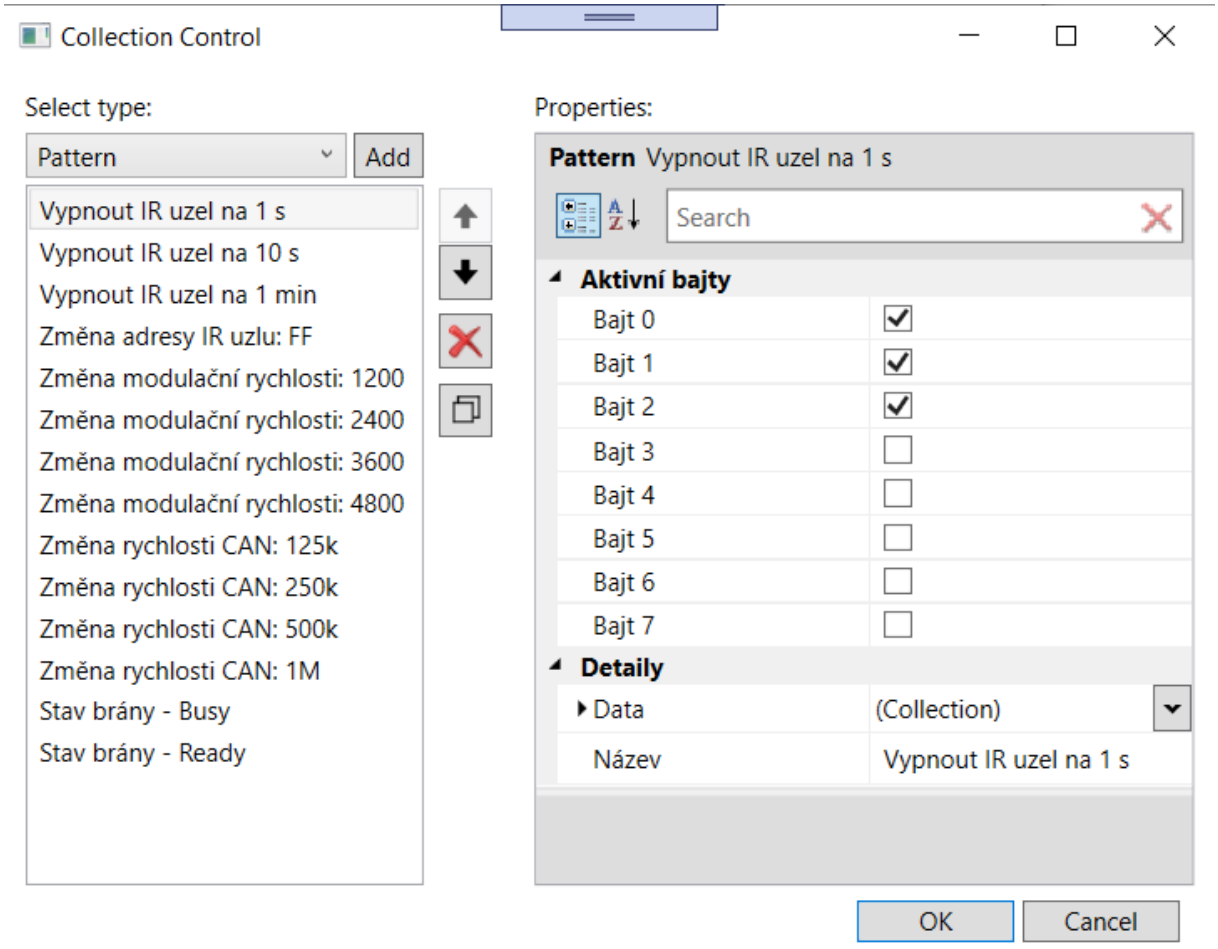
Zbylý obsah nastavení přísluší definici jednotlivých zpráv a vzorů. Po rozkliknutí kolonky Zprávy se zobrazí dialogové okno na obrázku 11. Zde je možné provádět definice nových zpráv, úpravy aktuálních zpráv, nebo taky mazání již vytvořených zpráv. Každá zpráva obsahuje svůj



Obrázek 11: Definice zprávy

jedinečný identifikátor, neboli ID, popis zprávy, který se zobrazí uživateli v hlavním okně aplikace, a soubor jednotlivých vzorů, které daná zpráva může obsahovat. Po rozkliknutí kolonky Vzory se otevře nové dialogové okno na obrázku 12.

Definice vzoru zprávy obsahuje název vzoru, který se zobrazí uživateli v hlavním okně aplikace, jednotlivá data zprávy a výběr aktivnívh bajtů, které chce uživatel ve zprávě odeslat. Podle aktivních bajtů zprávy se přepočítává i datová délka zprávy. Pokud jsou některá data ve zprávě vyplněna, ale nemají aktivní odpovídající bajty, daná data zůstanou prázdná. Jsou-li naopak aktivní bajty, které nemají vyplněná data, odešle se na tomto místě 0.



Obrázek 12: Definice vzoru zprávy

7 Testování

Při testování výsledného zařízení bylo potřeba otestovat celou cestu od komunikační sběrnice CAN po koncovou diagnostickou aplikaci na osobním počítači.

Jako první přišlo na řadu testování komunikačního uzlu na sběrnici CAN. Uzel převádí zprávy z formátu sběrnice CAN na zprávu ve formátu UART a naopak. Při testování byl uzel schopen převést libovolnou zprávu z jednoho formátu na druhý a odeslat zprávu správným směrem. To zvládá v čase desítek milisekund. Díky bufferu pro zprávy v obou směrech nedošlo ke ztrátám dat ani v případě, kdy zprávy přicházely krátkodobě ve vyšším intervalu, než je schopnost uzlu převádět zprávy.

Testování aplikace na osobním počítači probíhala metodou bílé skříňky. Aplikace vždy reagovala požadovaným způsobem v každé části programu. To zahrnuje i vypisování adekvátních chybových hlášek programu, nebo upozorňování uživatele na nesprávně zadané údaje.

Součástí aplikace je i funkční logovací systém. Do souboru log.txt se vypisují veškeré chyby, které v programu nastanou. Dále jsou v tomto souboru obsaženy taky informace o tom, co dělá uživatel v rámci programu, například proběhly-li změny v nastavení nebo že byla odeslána zpráva. Těchto zpráv je v programu pro logovací systém spousta. Aplikace využívá logovací systém NLog, což je volně šiřitelný logovací systém pro různé .NET platformy. Obsahem každého logu je vždy čas nastalé události, typ události a pak samotná zpráva o události. Typy událostí jsou podle důležitosti:

- Fatal
- Error
- Warn
- Info
- Debug
- Trace

Diagnostické zařízení tedy bez problémů zobrazuje uživateli veškeré dění na sběrnici CAN a umožňuje bez problémů odesílat příkazy z aplikace na sběrnici, které umožňují řízení koncového zařízení měřicí tlak. Každá zpráva obsahuje kontrolní součet CRC, jehož správnost kontroluje sběrnice CAN, komunikační uzel i aplikace na osobním počítači. Každý z těchto členů si spočítá CRC sám a zkontroluje ho s CRC obsaženým ve zprávě. správnost a celistvost jednotlivých zpráv je tímto způsobem zaručena.

8 Závěr

Cílem této práce bylo vytvoření diagnostického zařízení pro komunikační sběrnici CAN, která je součástí zařízení měřicí tlak. Toto zařízení se skládá ze dvou hlavním částí, a to ze speciálně upraveného komunikačního uzlu sběrnice CAN a z diagnostické aplikace zpracované ve WPF frameworku ve vývojovém prostředí Microsoft Visual Studio. To zahrnovalo použití tří programovacích jazyků. Jazyk C byl použit pro napsání programu pro komunikační uzel CAN. WPF framework pak vyžaduje použití dvou programovacích jazyků, a to jazyk C# pro obsluhu programu a jeho funkcí a jazyk XAML pro obsluhu uživatelského prostředí.

Tato práce byla rozdělena na teoretickou a praktickou část. Teoretická část se nachází v kapitolách 2, 3 a 4. Zabývala se především aktuálním stavem měřicího řetězce, možnými prostředky pro tvorbu diagnostické aplikace a obecnou definicí sběrnice CAN. Praktická část této práce obsahuje kapitoly 5, 6 a 7. Zde byla popsána realizace propojení měřicího řetězce s osobním počítačem, návrh a implementace uživatelského rozhraní a testování výsledného zařízení.

V kapitole 2 byl popsán aktuální stav měřicího řetězce. Bylo zde zobrazeno především blokové schéma měřicího řetězce spolu s vysvětlením principu každé jeho části.

Kapitola 3 se zabývala možnými prostředky pro tvorbu diagnostické aplikace na osobním počítači. To zahrnovalo výpis hlavních výhod a nevýhod programovacích jazyků vhodných pro tvorbu diagnostických aplikací. Vybrány zde byly jazyky C++, Java, C# a Python, protože se jedná o jedny z nejznámějších programovacích jazyků, které jsou dostupné zcela zdarma.

Poslední kapitolou teoretické části byla kapitola 4. Zabývala se stručnou historií a principem fungování obecné sběrnice CAN. V této kapitole bylo možné nalézt také CAN protokol, a to především jeho fyzickou a linkovou vrstvu.

Kapitola 5 byla první kapitolou praktické části této práce. Zabývala se propojením měřicího řetězce s osobním počítačem. V této kapitole byl popsán speciálně upravený komunikační uzel, komunikační protokol používaný diagnostickou aplikací na straně osobního počítače a také zde bylo možné nalézt ukázky kódu nahraných v mikrokontroléru tohoto komunikačního uzlu.

Samotná diagnostická aplikace na straně osobního počítače byla předvedena v kapitole 6. Součástí kapitoly byl taky popis WPF frameworku a postup při návrhu aplikace.

Poslední kapitolou praktické části této práce byla kapitola 7, která popisovala způsob testování vytvořeného diagnostického zařízení. Nakonec zde byl zmíněn způsob logování, jenž aplikace

používá, a taky způsob kontroly jednotlivých zpráv ve všech částech obvodu.

Na závěr bych rád zmínil, že celá práce je součástí většího projektu, který dále pokračuje a vytvořené zařízení může tedy být i následně upravováno či rozšiřováno. V budoucnu by bylo možné do aplikace přidat možnost přihlášení různých uživatelů s odlišnými přístupovými právy. Dále se dá stále zdokonalovat a rozšiřovat zobrazované informace o významu jednotlivých zpráv, nebo přidat možnost automatického vyhledávání připojeného komunikačního uzlu spolu s funkcí automatického připojení na správný sériový port.

Literatura

1. KAJZAR, Daniel. *Návrh designu elektroniky pro měření tlaku* [online]. Ostrava: Vysoká škola báňská - Technická univerzita Ostrava, 2019 [cit. 2019-12-13]. Dostupné z: https://dspace.vsb.cz/handle/10084/136159. Bakalářská práce.
2. KULA, Radim. *Návrh a implementace front-end pro konfiguraci zařízení měření tlaku* [online]. Ostrava: Vysoká škola báňská - Technická univerzita Ostrava, 2019 [cit. 2019-12-13]. Dostupné z: https://dspace.vsb.cz/handle/10084/136173. Diplomová práce.
3. MOWLAEE, Nader. *Top 10 Programming Languages for Engineers* [online]. 2019 [cit. 2019-12-15]. Dostupné z: https://interestingengineering.com/top-10-programming-languages-for-engineers.
4. BODOCKÝ, Martin. *Vznik a historie jazyka C++* [online]. 2006 [cit. 2019-12-15]. Dostupné z: http://programujte.com/clanek/2006061401-vznik-a-historie-jazyka-c/.
5. *Welcome back to C++ (Modern C++)* [online]. 2019 [cit. 2019-12-15]. Dostupné z: https://docs.microsoft.com/cs-cz/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=vs-2019.
6. *C++ Introduction* [online] [cit. 2019-12-15]. Dostupné z: https://www.w3schools.com/cpp/cpp_intro.asp.
7. FALTÝNEK, Lukáš. *Java – dnes při šálku dobré kávy* [online]. 2007 [cit. 2019-12-15]. Dostupné z: https://www.linuxexpres.cz/praxe/java-dnes-pri-salku-dobre-kavy.
8. GUPTA, Lokesh. *What is Java programming language?* [online] [cit. 2019-12-15]. Dostupné z: https://howtodoinjava.com/java/basics/what-is-java-programming-language/.
9. CHRISTIAN, Nagel. *Professional c# 7 and .net core 2.0*. Indiana: John Wiley, 2018. ISBN 978-1119449270.
10. O'REILLY, Tim. *Component-Oriented Versus Object-Oriented Programming* [online] [cit. 2019-12-15]. Dostupné z: https://www.oreilly.com/library/view/programming-net-components/0596102070/ch01s02.html.
11. KOSINA, Pavel. *Python - popis jazyka* [online]. 2005 [cit. 2019-12-18]. Dostupné z: http://programujte.com/clanek/1970010106-python-popis-jazyka/.
12. *Python Introduction* [online] [cit. 2019-12-18]. Dostupné z: https://www.w3schools.com/python/python_intro.asp.
13. POLÁK, Karel. *Sběrnice CAN* [online]. 2003 [cit. 2019-12-05]. Dostupné z: http://www.elektrorevue.cz/clanky/03021/index.html.
14. *CAN - Controller Area Network* [online] [cit. 2019-12-07]. Dostupné z: http://www.canlab.cz/cs/can_bus.

15. KALČÍK, Martin. *Implementace sběrnice CAN s ohledem na EMC* [online]. Plzeň: ZÁPADOČESKÁ UNIVERZITA V PLZNI, 2013 [cit. 2019-12-13]. Dostupné z: `https://dspace5.zcu.cz/handle/11025/8247`. Bakalářská práce.

16. *CAN protocol tutorial* [online] [cit. 2019-12-05]. Dostupné z: `https://www.kvaser.com/can-protocol-tutorial/`.

17. CAMPBELL, Scott. *BASICS OF UART COMMUNICATION* [online] [cit. 2019-12-14]. Dostupné z: `http://www.circuitbasics.com/basics-uart-communication/`.

18. SIMPSON, W. *PPP in HDLC-like Framing* [online]. 1994 [cit. 2019-12-14]. Dostupné z: `https://whois.smartweb.cz/rfc/rfc1662/`.

19. DAJBYCH, Václav. *mvvm: model-view-viewmodel* [online]. 2009 [cit. 2020-03-26]. Dostupné z: `https://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel`.

20. *What is XAML* [online] [cit. 2020-03-26]. Dostupné z: `https://www.wpf-tutorial.com/xaml/what-is-xaml/`.

A Standardní formát rámce CAN zprávy

Tabulka 2: Standardní formát rámce CAN zprávy bez vkládaných bitů

Hlavní pole rámce	Název parametru	Bitová délka	Význam
Startovací pole	Startovací bit	1	Začátek přenášeného rámce.
Arbitráž	Identifikátor (ID)	11	Jedinečný identifikátor určující prioritu a obsah zprávy.
	RTR	1	Žádost o vzdálený přenos (remote transmission request) musí být dominantní pro datový rámec a recesivní pro vzdálený rámec.
Kontrolní pole	IDE	1	Prodloužené označení (Identifier extension bit) musí být recesivní pro rozšířený formát a dominantní pro standardní formát.
	Rezervovaný bit	1	Musí být dominantní.
	Datová délka	4	Počet bajtů datového pole.
Datové pole	Datové pole	0–64	Data k odeslání (0–8 B).
Pole CRC	CRC	15	Bezpečnostní kontrolní součet dat pro detekci bitových chyb.
	Oddělovač CRC	1	(CRC delimiter) Rezervovaný bit, musí být recesivní. Je-li dominantní, sběrnice hlásí CRC chybu.
Pole ACK	ACK	1	Potvrzení přijetí zprávy (Acknowledgement bit). Vysílač vyšle bit jako recesivní a jakýkoliv přijímač tento bit označí za dominantní. Tím vysílač pozná, že zprávu někdo přijal.
	Oddělovač ACK	1	(ACK delimiter) Rezervovaný bit, musí být recesivní. Je-li dominantní, sběrnice hlásí ACK chybu.
Koncové pole	Koncové pole	7	Konec přenášeného rámce.

B Rozšířený formát rámce CAN zprávy

Tabulka 3: Rozšířený formát rámce CAN zprávy bez vkládaných bitů

Hlavní pole rámce	Název parametru	Bitová délka	Význam
Startovací pole	Startovací bit	1	Začátek přenášeného rámce.
Arbitráž	Identifikátor A	11	První část jedinečného identifikátoru určujícího prioritu a obsah zprávy.
	SRR	1	Musí být recesivní (Substitute remote request).
	IDE	1	Prodloužené označení (Identifier extension bit) musí být recesivní pro rozšířený formát a dominantní pro standardní formát.
	Identifikátor B	18	Druhá část jedinečného identifikátoru určujícího prioritu a obsah zprávy.
	RTR	1	Žádost o vzdálený přenos (remote transmission request) musí být dominantní pro datový rámec a recesivní pro vzdálený rámec.
Kontrolní pole	Rezervované bity	2	Musí být dominantní.
	Datová délka	4	Počet bajtů datového pole.
Datové pole	Datové pole	0–64	Data k odeslání (0–8 B).
Pole CRC	CRC	15	Bezpečnostní kontrolní součet dat pro detekci bitových chyb.
	Oddělovač CRC	1	(CRC delimiter) Rezervovaný bit, musí být recesivní. Je-li dominantní, sběrnice hlásí CRC chybu.
Pole ACK	ACK	1	Potvrzení přijetí zprávy (Acknowledgement bit). Vysílač vyšle bit jako recesivní a jakýkoliv přijímač tento bit označí za dominantní. Tím vysílač pozná, že zprávu někdo přijal.
	Oddělovač ACK	1	(ACK delimiter) Rezervovaný bit, musí být recesivní. Je-li dominantní, sběrnice hlásí ACK chybu.
Koncové pole	Koncové pole	7	Konec přenášeného rámce.

C Funkce v jazyce C pro převod komunikace z UART na CAN

```
CANMessage UARTToCAN(unsigned char *UARTData)
{
    CANMessage packet;
    int position = 1;

    packet.type = UARTData[position];                    // Type
    position += 4;

    packet.can_packet.ID = 0;
    for(i=0;i<4;i++) {                                   // ID
        if(UARTData[position] == 0x7D) {
            position++;
            if(UARTData[position] == 0x5E) {
                packet.can_packet.ID += 0x7E << (24 - i*8);
                position++;
            }
            else if(UARTData[position] == 0x5D) {
                packet.can_packet.ID += 0x7D << (24 - i*8);
                position++;
            }
        }
        else {
            packet.can_packet.ID += (unsigned long)(UARTData[position]) << (24 -
                 i*8);
            position++;
        }
    }

    packet.can_packet.dataLength = UARTData[position]; // Data length
    position += 4;

    for(i=0;i<packet.can_packet.dataLength;i++) {      // Data
        if(UARTData[position] == 0x7D) {
            position++;
            if(UARTData[position] == 0x5E) {
                packet.can_packet.data[i] = 0x7E;
                position++;
```

```
            }
            else if(UARTData[position] == 0x5D) {
                packet.can_packet.data[i] = 0x7D;
                position++;
            }
        }
        else {
            packet.can_packet.data[i] = UARTData[position];
            position++;
        }
    }

    for(i=0;i<4;i++) {                                  // CRC
        packet.crc[i] = UARTData[position];
        position++;
    }

    return packet;
}
```

Výpis 7: Funkce v jazyce C pro převod komunikace z UART na CAN

D Funkce v jazyce C pro převod komunikace z CAN na UART

```
void CANToUART(CANPacket packet, unsigned char *UARTData)
{
    sizeOfMessage = 0;
    unsigned int crc = crc32(packet.data);

    UARTData[sizeOfMessage] = 0x7E;                              // Start
    sizeOfMessage++;

    UARTData[sizeOfMessage] = 0x02;                              // Type
    sizeOfMessage++;
    UARTData[sizeOfMessage] = UARTData[sizeOfMessage] = UARTData[sizeOfMessage
        + 1] = 0x00;
    sizeOfMessage += 3;                                          // rsvd

    for(i=0;i<4;i++) {                                           // ID
        if(((unsigned char)(packet.ID >> (24 - i*8))) == 0x7E) {
            UARTData[sizeOfMessage] = 0x7D;
            sizeOfMessage++;
            UARTData[sizeOfMessage] = 0x5E;
            sizeOfMessage++;
        }
        else if(((unsigned char)(packet.ID >> (24 - i*8))) == 0x7D) {
            UARTData[sizeOfMessage] = 0x7D;
            sizeOfMessage++;
            UARTData[sizeOfMessage] = 0x5D;
            sizeOfMessage++;
        }
        else {
            UARTData[sizeOfMessage] = ((unsigned char)(packet.ID >> (24 - i*8)))
                ;
            sizeOfMessage++;
        }
    }

    UARTData[sizeOfMessage] = packet.dataLength;                 // Data length
    sizeOfMessage++;
```

```
    UARTData[sizeOfMessage] = UARTData[sizeOfMessage] = UARTData[sizeOfMessage
        + 1] = 0x00;
    sizeOfMessage += 3;                                         // rsvd

    for(i=0;i<packet.dataLength;i++) {                          // Data
        if(packet.data[i] == 0x7E) {
            UARTData[sizeOfMessage] = 0x7D;
            sizeOfMessage++;
            UARTData[sizeOfMessage] = 0x5E;
            sizeOfMessage++;
        }
        else if(packet.data[i] == 0x7D) {
            UARTData[sizeOfMessage] = 0x7D;
            sizeOfMessage++;
            UARTData[sizeOfMessage] = 0x5D;
            sizeOfMessage++;
        }
        else {
            UARTData[sizeOfMessage] = packet.data[i];
            sizeOfMessage++;
        }
    }

    for(i=0;i<4;i++) {                                          // CRC
        UARTData[sizeOfMessage] = crc << (24 - i*8);
        sizeOfMessage++;
    }

    UARTData[sizeOfMessage] = 0x7E;                             // End
    sizeOfMessage++;
}
```

Výpis 8: Funkce v jazyce C pro převod komunikace z CAN na UART